Chapter 8

Exceptional Control Flow

From the time you first apply power to a processor until the time you shut it off, the program counter assumes a sequence of values

$$a_0, a_1, \ldots, a_{n-1}.$$

where each a_k is the address of some corresponding instruction I_k. Each transition from a_k to a_{k+1} is called a *control transfer*. A sequence of such control transfers is called the *flow of control*, or *control flow* of the processor.

The simplest kind of control flow is a smooth sequence where each I_k and I_{k+1} are adjacent in memory. Typically, abrupt changes to this smooth flow, where I_{k+1} is not adjacent to I_k, are caused by familiar program instructions such as jumps, calls, and returns. Such instructions are necessary mechanisms that allow programs to react to changes in internal program state represented by program variables.

But systems must also be able to react to changes in system state that are not captured by internal program variables and are not necessarily related to the execution of the program. For example, a hardware timer goes off at regular intervals and must be dealt with. Packets arrive at the network adapter and must be stored in memory. Programs request data from a disk and then sleep until they are notified that the data are ready. Parent processes that create child processes must be notified when their children terminate.

Modern systems react to these situations by making abrupt changes in the control flow. We refer to these abrupt changes in general as *exceptional control flow*. Exceptional control flow occurs at all levels of a computer system. For example, at the hardware level, events detected by the hardware trigger abrupt control transfers to exception handlers. At the operating systems level, the kernel transfers control from one user process to another via context switches. At the application level, a process can send a *Unix signal* to another process that abruptly transfers control to a signal handler in the recipient. An individual program can react to errors by sidestepping the usual stack discipline and making nonlocal jumps to arbitrary locations in other functions (similar to the *exceptions* supported by C++ and Java).

This chapter describes these various forms of exceptional control, and shows you how to use them in your C programs. The techniques you will learn about — creating processes, reaping terminated processes, sending and receiving signals, making non-local jumps — are the foundation of important programs such as Unix shells (Problem 8.20) and Web servers (Chapter 12).

8.1 Exceptions

Exceptions are a form of exceptional control flow that are implemented partly by the hardware and partly by the operating system. Because they are partly implemented in hardware, the details vary from system to system. However, the basic ideas are the same for every system. Our aim in this section is to give you a general understanding of exceptions and exception handling, and to help demystify what is often a confusing aspect of modern computer systems.

An *exception* is an abrupt change in the control flow in response to some change in the processor's state. Figure 8.1 shows the basic idea.

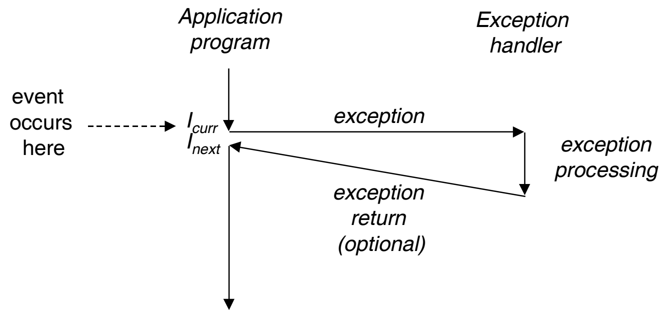


Figure 8.1: **Anatomy of an exception.** A change in the processor's state (event) triggers an abrupt control transfer (an exception) from the application program to an exception handler. After it finishes processing, the handler either returns control to the interrupted program or aborts.

In the figure, the processor is executing some current instruction I_{curr} when a significant change in the processor's *state* occurs. The state is encoded in various bits and signals inside the processor. The change in state is known as an *event*. The event might be directly related to the execution of the current instruction. For example, a virtual memory page fault occurs, an arithmetic overflow occurs, or an instruction attempts a divide by zero. On the other hand, the event might be unrelated to the execution of the current instruction. For example, a system timer goes off or an I/O request completes.

In any case, when the processor detects that the event has occurred, it makes an indirect procedure call (the exception), through a jump table called an *exception table*, to an operating system subroutine (the *exception handler*) that is specifically designed to process this particular kind of event.

When the exception handler finishes processing, one of three things happens, depending on the type of event that caused the exception:

1. The handler returns control to the current instruction I_{curr}, the instruction that was executing when the event occurred.
2. The handler returns control to I_{next}, the instruction that would have executed next had the exception not occurred.
3. The handler aborts the interrupted program.

Section 8.1.2 says more about these possibilities.

8.1.1 Exception Handling

Exceptions can be difficult to understand because handling them involves close cooperation between hardware and software. It is easy to get confused about which component performs which task. Let's look at the division of labor between hardware and software in more detail.

Each type of possible exception in a system is assigned a unique non-negative integer *exception number*. Some of these numbers are assigned by the designers of the processor. Other numbers are assigned by the designers of the operating system *kernel* (the memory-resident part of the operating system). Examples of the former include divide by zero, page faults, memory access violations, breakpoints, and arithmetic overflows. Examples of the latter include system calls and signals from external I/O devices.

At system boot time (when the computer is reset or powered on) the operating system allocates and initializes a jump table called an *exception table*, so that entry k contains the address of the handler for exception k. Figure 8.2 shows the format of an exception table.

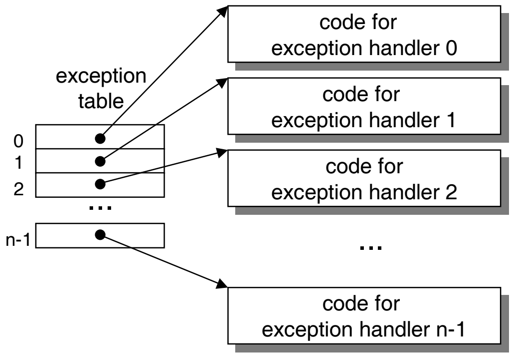

exception table
0
1
2
...
n-1
code for exception handler 0
code for exception handler 1
code for exception handler 2
...
code for exception handler n-1

Figure 8.2: **Exception table.** The exception table is a jump table where entry k contains the address of the handler code for exception k.

At runtime (when the system is executing some program), the processor detects that an event has occurred and determines the corresponding exception number k. The processor then triggers the exception by making an indirect procedure call, through entry k of the exception table, to the corresponding handler. Figure 8.3 shows how the processor uses the exception table to form the address of the appropriate exception handler. The exception number is an index into the exception table, whose starting address is contained in a special CPU register called the *exception table base register*.

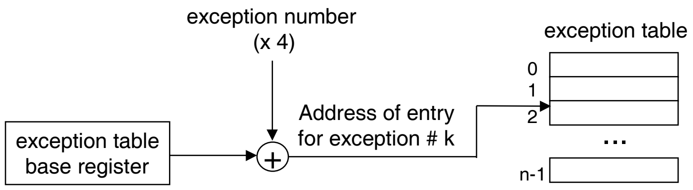


Figure 8.3: **Generating the address of an exception handler.** The exception number is an index into the exception table.

An exception is akin to a procedure call, but with some important differences.

- As with a procedure call, the processor pushes a return address on the stack before branching to the handler. However, depending on the class of exception, the return address is either the current instruction (the instruction that was executing when the event occurred) or the next instruction (the instruction that would have executed after the current instruction had the event not occurred).
- The processor also pushes some additional processor state onto the stack that will be necessary to restart the interrupted program when the handler returns. For example, an IA32 system pushes the EFLAGS register containing, among other things, the current condition codes, onto the stack.
- If control is being transferred from a user program to the kernel, all of the above items are pushed on the kernel's stack rather than the user's stack.
- Exception handlers run in *kernel mode* (Section 8.2.3, which means they have complete access to all system resources.

Once the hardware triggers the exception, the rest of the work is done in software by the exception handler. After the handler has processed the event, it optionally returns to the interrupted program by executing a special "return from interrupt" instruction, which pops the appropriate state back into the processor's control and data registers, restores the state to *user mode* (Section 8.2.3) if the exeption interrupted a user program, and then returns control to the interrupted program.

8.1.2 Classes of Exceptions

Exceptions can be divided into four classes: *interrupts*, *traps*, *faults*, and *aborts*. Figure 8.4 summarizes the attributes of these classes.

Class	Cause	Async/Sync	Return behavior
Interrupt	Signal from I/O device	Async	always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4: **Classes of exceptions.** Asynchronous exceptions occur as a result of events external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

Interrupts

Interrupts occur *asynchronously* as a result of signals from I/O devices that are external to the processor. Hardware interrupts are asynchronous in the sense that they are not caused by the execution of any particular instruction. Exception handlers for hardware interrupts are often called *interrupt handlers*.

Figure 8.5 summarizes the processing for an interrupt. I/O devices such as network adapters, disk controllers, and timer chips trigger interrupts by signalling a pin on the processor chip and placing the exception number on the system bus that identifies the device that caused the interrupt.

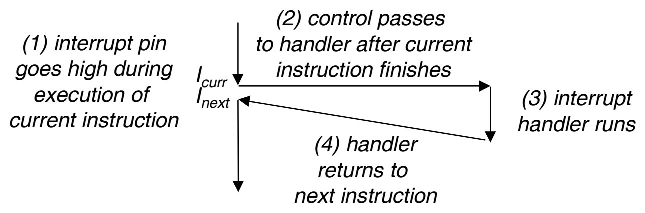

(1) interrupt pin goes high during execution of current instruction
(2) control passes to handler after current instruction finishes
I_{curr}
I_{next}
(3) interrupt handler runs
(4) handler returns to next instruction

Figure 8.5: **Interrupt handling.** The interrupt handler returns control to the next instruction in the application program's control flow.

After the current instruction finishes executing, the processor notices that the interrupt pin has gone high, reads the exception number from the system bus, and then calls the appropriate interrupt handler. When the handler returns, it returns control to the next instruction (i.e., the instruction that would have followed the current instruction in the control flow had the interrupt not occurred). The effect is that the program continues executing as though the interrupt had never happened.

The remaining classes of exceptions (traps, faults, and aborts) occur *synchronously* as a result of executing the current instruction. We refer to this instruction as the *faulting instruction*.

Traps

Traps are *intentional* exceptions that occur as a result of executing an instruction. Like interrupt handlers, trap handlers return control to the next instruction. The most important use of traps is to provide a procedure-like interface between user programs and the kernel known as a *system call*.

User programs often need to request services from the kernel such as reading a file (`read`), creating a new process (`fork`), loading a new program (`execve`), or terminating the current process (`exit`). To allow controlled access to such kernel services, processors provide a special "`syscall` n" instruction that user programs can execute when they want to request service n. Executing the `syscall` instruction causes a trap to an exception handler that decodes the argument and calls the appropriate kernel routine. Figure 8.6 summarizes the processing for a system call. From a programmer's perspective, a system call is identical

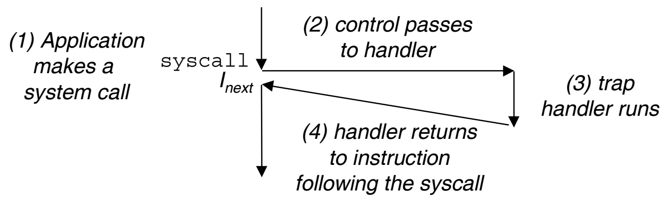

(1) Application makes a system call
syscall
I_{next}
(2) control passes to handler
(3) trap handler runs
(4) handler returns to instruction following the syscall

Figure 8.6: **Trap handling.** The trap handler returns control to the next instruction in the application program's control flow.

to a regular function call. However, their implementations are quite different. Regular functions run in *user mode*, which restricts the types of instructions they can execute, and they access the same stack as the calling function. A system call runs in *kernel mode*, which allows it to execute instructions, and accesses a stack defined in the kernel. Section 8.2.3 discusses user and kernel modes in more detail.

Faults

Faults result from error conditions that a handler might be able to correct. When a fault occurs, the processor transfers control to the fault handler. If the handler is able to correct the error condition, it returns control to the faulting instruction, thereby reexecuting it. Otherwise, the handler returns to an `abort` routine in the kernel that terminates the application program that caused the fault. Figure 8.7 summarizes the processing for a fault.

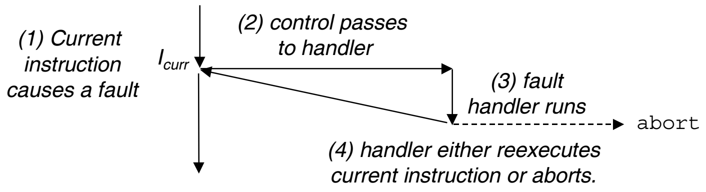

(1) Current instruction causes a fault
I_{curr}
(2) control passes to handler
(3) fault handler runs
abort
(4) handler either reexecutes current instruction or aborts.

Figure 8.7: **Fault handling.** Depending on the whether the fault can be repaired or not, the fault handler either re-executes the faulting instruction or aborts.

A classic example of a fault is the page fault exception, which occurs when an instruction references a virtual address whose corresponding physical page is not resident in memory and must be retrieved from disk. As we will see in Chapter 10, a page is contiguous block (typically 4 KB) of virtual memory. The page fault handler loads the appropriate page from disk and then returns control to the instruction that caused the fault. When the instruction executes again, the appropriate physical page is resident in memory and the instruction is able to run to completion without faulting.

Aborts

Aborts result from unrecoverable fatal errors, typically hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted. Abort handlers never return control to the application program. As shown in Figure 8.8, the handler returns control to an `abort` routine that terminates the application program.

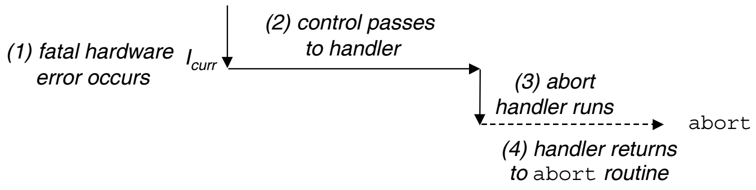

(1) fatal hardware error occurs
I_{curr}
(2) control passes to handler
(3) abort handler runs
abort
(4) handler returns to abort routine

Figure 8.8: **Abort handling.** The abort handler passes control to a kernel `abort` routine that terminates the application program.

8.1.3 Exceptions in Intel Processors

To help make things more concrete, let's look at some of the exceptions defined for Intel systems. A Pentium system can have up to 256 different exception types. Numbers in the range 0 to 31 correspond to exceptions that are defined by the Pentium architecture, and thus are identical for any Pentium-class system. Numbers in the range 32 to 255 correspond to interrupts and traps that are defined by the operating system. Figure 8.9 shows a few examples.

Exception Number	Description	Exception Class
0	divide error	fault
13	general protection fault	fault
14	page fault	fault
18	machine check	abort
32–127	OS-defined exceptions	interrupt or trap
128 (`0x80`)	system call	trap
129–255	OS-defined exceptions	interrupt or trap

Figure 8.9: **Examples of exceptions in Pentium systems.**

A divide error (exception 0) occurs when an application attempts to divide by zero, or when the result of a divide instruction is too big for the destination operand. Unix does not attempt to recover from divide errors, opting instead to abort the program. Unix shells typically report divide errors as "Floating exceptions".

The infamous general protection fault (exception 13) occurs for many reasons, usually because a program references an undefined area of virtual memory, or because the program attempts to write to a read-only text segment. Unix does not attempt to recover from this fault. Unix shells typically report general protection faults as "Segmentation faults".

A page fault (exception 14) is an example of an exception where the faulting instruction is restarted. The handler maps the appropriate page of physical memory on disk into a page of virtual memory, and then restarts the faulting instruction. We will see how this works in detail in Chapter 10.

A machine check (exception 18) occurs as a result of a fatal hardware error that is detected during the execution of the faulting instruction. Machine check handlers never return control to the application program.

System calls are provided on IA32 systems via a trapping instruction called `INT` n, where n can be the index of any of the 256 entries in the exception table. Historically, systems calls are provided through exception 128 (`0x80`).

Aside: A note on terminology.
The terminology for the various classes of exceptions varies from system to system. Processor macro-architecture specifications often distinguish between asynchronous "interrupts" and synchronous "exceptions", yet provide no umbrella term to refer to these very similar concepts. To avoid having to constantly refer to "exceptions and interrupts" and "exceptions or interrupts", we use the word "exception" as the general term and distinguish between asynchronous exceptions (interrupts) and synchronous exceptions (traps, faults, and aborts) only when it is appropriate. As we have noted, the basic ideas are the same for every system, but you should be aware that some manufacturers' manuals use the word "exception" to refer only to those changes in control flow caused by synchronous events. **End Aside.**

8.2 Processes

Exceptions provide the basic building blocks that allow the operating system to provide the notion of a *process*, one of the most profound and successful ideas in computer science.

When we run a program on a modern system, we are presented with the illusion that our program is the only one currently running in the system. Our program appears to have exclusive use of both the processor and the memory. The processor appears to execute the instructions in our program, one after the other, without interruption. And the code and data of our program appear to be the only objects in the system's memory. These illusions are provided to us by the notion of a process.

The classic definition of a process is *an instance of a program in execution*. Each program in the system runs in the *context* of some process. The context consists of the state that the program needs to run correctly. This state includes the program's code and data stored in memory, its stack, the contents of its general-purpose registers, its program counter, environment variables, and the set of open file descriptors.

Each time a user runs a program by typing the name of an executable object file to the shell, the shell creates a new process and then runs the executable object file in the context of this new process. Application programs can also create new processes and run either their own code or other applications in the context of the new process.

A detailed discussion of how operating systems implement processes is beyond our scope. Instead we will focus on the key abstractions that a process provides to the application:

- An independent *logical control flow* that provides the illusion that our program has exclusive use of the processor.
- A private address space that provides the illusion that our program has exclusive use of the memory system.

Let's look more closely at these abstractions.

8.2.1 Logical Control Flow

A process provides each program with the illusion that it has exclusive use of the processor, even though many other programs are typically running on the system. If we were to use a debugger to single step the execution of our program, we would observe a series of program counter (PC) values that corresponded exclusively to instructions contained in our program's executable object file or in shared objects linked into our program dynamically at run time. This sequence of PC values is known as a *logical control flow*.

Consider a system that runs three processes, as shown in Figure 8.10. The single physical control flow of the processor is partitioned into three *logical flows*, one for each process. Each vertical line represents a portion of the logical flow for a process. In the example, process A runs for a while, followed by B, which runs to completion. Then C runs for awhile, followed by A, which runs to completion. Finally, C is able to run to completion.

The key point in Figure 8.10 is that processes take turns using the processor. Each process executes a portion of its flow and then is *preempted* (temporarily suspended) while other processes take their turns. To

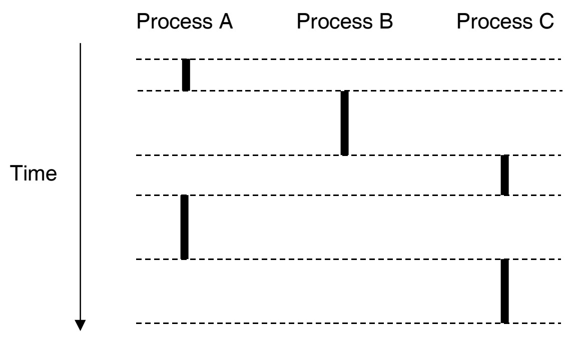


Figure 8.10: **Logical control flows.** Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar represents a portion of the logical control flow for a process.

a program running in the context of one of these processes, it appears to have exclusive use of the processor. The only evidence to the contrary is that if we were to precisely measure the elapsed time of each instruction (see Chapter 9), we would notice that the CPU appears to periodically stall between the execution of some of the instructions in our program. However, each time the processor stalls, it subsequently resumes execution of our program without any change to the contents of the program's memory locations or registers.

In general, each logical flow is independent of any other flow in the sense that the logical flows associated with different processes do not affect the states of any other processes. The only exception to this rule occurs when processes use interprocess communication (IPC) mechanisms such as pipes, sockets, shared memory, and semaphores to explicitly interact with each other.

Any process whose logical flow overlaps in time with another flow is called a *concurrent process*, and the two processes are said to run *concurrently*. For example, in Figure 8.10, processes A and B run concurrently, as do A and C. On the other hand, B and C do not run concurrently because the last instruction of B executes before the first instruction of C.

The notion of processes taking turns with other processes is known as *multitasking*. Each time period that a process executes a portion of its flow is called a *time slice*. Thus, multitasking is also referred to as *time slicing*.

8.2.2 Private Address Space

A process also provides each program with the illusion that it has exclusive use of the system's address space. On a machine with n-bit addresses, the *address space* is the set of 2^n possible addresses, $0, 1, \ldots, 2^n - 1$. A process provides each program with its own *private address space*. This space is private in the sense that a byte of memory associated with a particular address in the space cannot in general be read or written by any other process.

Although the contents of the memory associated with each private address space is different in general, each such space has the same general organization. For example, Figure 8.11 shows the organization of the address space for a Linux process. The bottom three-fourths of the address space is reserved for the user program, with the usual text, data, heap, and stack segments. The top quarter of the address space is reserved for the kernel. This portion of the address space contains the code, data, and stack that the kernel

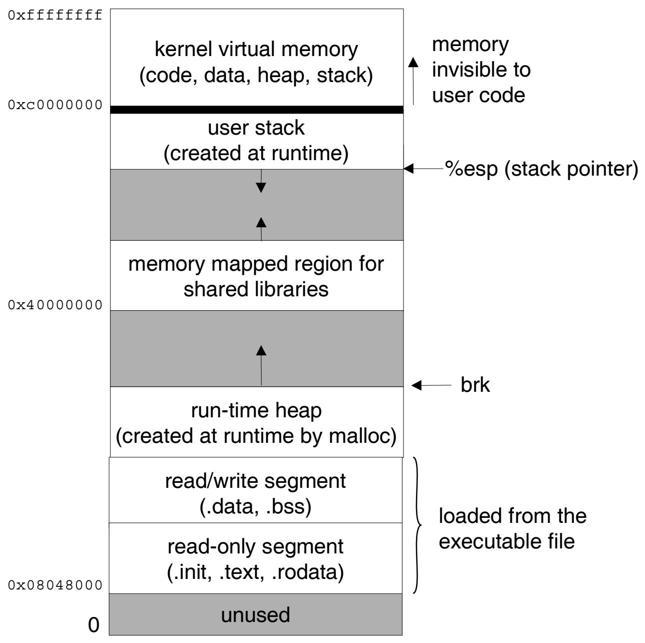


Figure 8.11: **Process address space.**

uses when it executes instructions on behalf of the process (e.g., when the application program executes a system call).

8.2.3 User and Kernel Modes

In order for the operating system kernel to provide an airtight process abstraction, the processor must provide a mechanism that restricts the instructions that an application can execute, as well as the portions of the address space that it can access.

Processors typically provide this capability with a *mode bit* in some control register that characterizes the privileges that the process currently enjoys. When the mode bit is set, the process is running in *kernel mode* (sometimes called *supervisor mode*). A process running in kernel mode can execute any instruction in the instruction set and access any memory location in the system.

When the mode bit is not set, the process is running in *user mode*. A process in user mode is not allowed to execute *privileged instructions* that do things such as halt the processor, change the mode bit, or initiate an I/O operation. Nor is it allowed to directly reference code or data in the kernel area of the address space. Any such attempt results in a fatal protection fault. Instead, user programs must access kernel code and data indirectly via the system call interface.

A process running application code is initially in user mode. The only way for the process to change from user mode to kernel mode is via an exception such as an interrupt, a fault, or a trapping system call. When the exception occurs, and control passes to the exception handler, the processor changes the mode from user mode to kernel mode. The handler runs in kernel mode. When it returns to the application code, the processor changes the mode from kernel mode back to user mode.

Linux and Solaris provides a clever mechanism, called the `/proc` filesystem, that allows user mode processes to access the contents of kernel data structures. The `/proc` filesystem exports the contents of many kernel data structures as a hierarchy of ASCII files that can read by user programs. For example, you can use the Linux `proc` filesystem to find out general system attributes such as CPU type (`/proc/cpuinfo`), or the memory segments used by a particular process (`/proc/<process id>/maps`).

8.2.4 Context Switches

The operating system kernel implements multitasking using a higher-level form of exceptional control flow known as a *context switch*. The context switch mechanism is built on top of the lower-level exception mechanism that we discussed in Section 8.1.

The kernel maintains a *context* for each process. The context is the state that the kernel needs to restart a preempted process. It consists of the values of objects such as the general-purpose registers, the floating-point registers, the program counter, user's stack, status registers, kernel's stack, and various kernel data structures such as a *page table* that characterizes the address space, a *process table* that contains information about the current process, and a *file table* that contains information about the files that the process has opened.

At certain points during the execution of a process, the kernel can decide to preempt the current process and restart a previously preempted process. This decision is known as *scheduling*, and is handled by a part of the kernel called the *scheduler*. When the kernel selects a new process to run, we say that the kernel has *scheduled* that process. After the kernel has scheduled a new process to run, it preempts the current process and transfers control to the new process using a mechanism called a *context switch* that (1) saves the context of the current process, (2) restores the saved context of some previously preempted process, and (3) passes control to this newly restored process.

A context switch can occur while the kernel is executing a system call on behalf of the user. If the system call blocks because it is waiting for some event to occur, then the kernel can put the current process to sleep and switch to another process. For example, if a `read` system call requires a disk access, the kernel can opt to perform a context switch and run another process instead of waiting for the data to arrive from the disk. Another example is the `sleep` system call, which is an explicit request to put the calling process to sleep. In general, even if a system call does not block, the kernel can decide to perform a context switch rather than return control to the calling process.

A context switch can also occur as a result of an interrupt. For example, all systems have some mechanism for generating periodic timer interrupts, typically every 1 ms or 10 ms. Each time a timer interrupt occurs, the kernel can decide that the current process has run long enough and switch to a new process.

Figure 8.12 shows an example of context switching between a pair of processes A and B. In this example, initially process A is running in user mode until it traps to the kernel by executing a `read` system call. The trap handler in the kernel requests a DMA transfer from the disk controller and arranges for the disk to interrupt the processor after the disk controller has finished transferring the data from disk to memory.

The disk will take a relatively long time to fetch the data (on the order of tens of milliseconds), so instead of waiting and doing nothing in the interim, the kernel performs a context switch from process A to B. Note that before the switch, the kernel is executing instructions in user mode on behalf of process A. During the

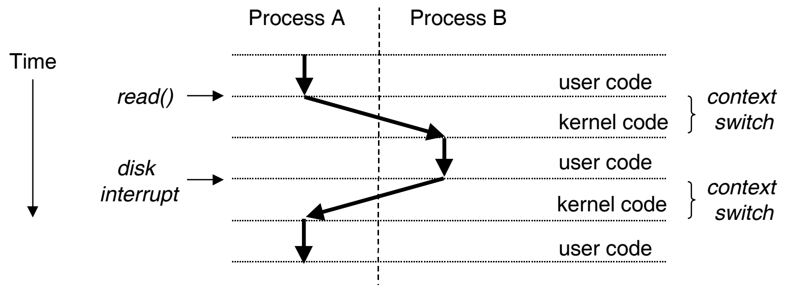


Figure 8.12: **Anatomy of a context switch.**

first part of the switch, the kernel is executing instructions in kernel mode on behalf of process A. Then at some point it begins executing instructions (still in kernel mode) on behalf of process B. And after the switch, the kernel is executing instructions in user mode on behalf of process B.

Process B then runs for a while in user mode until the disk sends an interrupt to signal that data has been transferred from disk to memory. The kernel decides that process B has run long enough and performs a context switch from process B to A, returning control in process A to the instruction immediately following the `read` system call. Process A continues to run until the next exception occurs, and so on.

8.3 System Calls and Error Handling

Unix systems provide a variety of systems calls that application programs use when they want to request services from the kernel such as reading a file or creating a new process. For example, Linux provides about 160 system calls. Typing "`man syscalls`" will give you the complete list.

C programs can invoke any system call directly by using the `_syscall` macro described in "`man 2 intro`". However, it is usually neither necessary nor desirable to invoke system calls directly. The standard C library provides a set of convenient wrapper functions for the most frequently used system calls. The wrapper functions package up the arguments, trap to the kernel with the appropriate system call, and then pass the return status of the system call back to the calling program. In our discussion in the following sections, we will refer to system calls and their associated wrapper functions interchangeably as *system-level functions*.

When Unix system-level functions encounter an error, they typically return -1 and set the global integer variable `errno` to indicate what went wrong. Programmers should *always* check for errors, but unfortunately, many skip error checking because it bloats the code and makes it harder to read. For example, here is how we might check for errors when we call the Unix `fork` function:

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(0);
}
```

The `strerror` function returns a text string that describes the error associated with a particular value of `errno`. We can simplify this code somewhat by defining the following *error-reporting function*:

```
void unix_error(char *msg) /* unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
```

Given this function, our call to `fork` reduces from four lines to two lines:

```
    if ((pid = fork()) < 0)
        unix_error("fork error");
```

We can simplify our code even further by using a *error-handling wrappers*. For a given base function `foo`, we define a wrapper function `Foo` with identical arguments, but with the first letter of the name capitalized. The wrapper calls the base function, checks for errors and terminates if there are any problems. For example, here is the error-handling wrapper for the `fork` function:

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

Given this wrapper, our call to `fork` shrinks to a single compact line:

```
    pid = Fork();
```

We will use error-handling wrappers throughout the remainder of this book. They allow us to keep our code examples concise, without giving you the mistaken impression that it is permissible to ignore error-checking. Note that when we discuss system-level functions in the text, we will always refer to them by their lower-case base names, rather than by their upper-case wrapper names.

See Appendix A for a discussion of Unix error-handling and the error-handling wrappers used throughout the book. The wrappers are defined in a file called `csapp.c` and their prototypes are defined in a header file called `csapp.h`. For your reference, Appendix A provides the sources for these files.

8.4 Process Control

Unix provides a number of system calls for manipulating processes from C programs. This section describes the important functions and gives examples of how they are used.

8.4.1 Obtaining Process ID's

Each process has a unique positive (non-zero) *process ID (PID)*. The `getpid` function returns the PID of the calling process. The `getppid` function returns the PID of its *parent* (i.e., the process that created the calling process).

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);
```
returns: PID of either the caller or the parent

The `getpid` and `getppid` routines return an integer value of type `pid_t`, which on Linux systems is defined in `types.h` as an `int`.

8.4.2 Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states:

- *Running.* The process is either executing on the CPU, or is waiting to be executed and will eventually be scheduled.
- *Stopped.* The execution of the process is *suspended* and will not be scheduled. A process stops as a result of receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal, and it remains stopped until it receives a SIGCONT signal, at which point is becomes running again. (A *signal* is a form of software interrupt that is described in detail in Section 8.5.)
- *Terminated.* The process is stopped permanently. A process becomes terminated for one of three reasons: (1) receiving a signal whose default action is to terminate the process; (2) returning from the main routine; or (3) calling the `exit` function:

```
#include <stdlib.h>

void exit(int status);
```
this function does not return

The `exit` function terminates the process with an *exit status* of `status`. (The other way to set the exit status is to return an integer value from the main routine.)

A *parent process* creates a new running *child process* by calling the `fork` function.

```
#include <unistd.h>
#include <sys/types.h>

pid_t fork(void);
```

returns: 0 to child, PID of child to parent, -1 on error

The newly created child process is almost, but not quite, identical to the parent. The child gets an identical (but separate) copy of the parent's user-level virtual address space, including the text, data, and bss segments, heap, and user stack. The child also gets identical copies of any of the parent's open file descriptors, which means the child can read and write any files that were open in the parent when it called `fork`. The most significant difference between the parent and the newly created child is that they have different PIDs.

The `fork` function is interesting (and often confusing) because it is called *once* but it returns *twice*: once in the calling process (the parent), and once in the newly created child process. In the parent, `fork` returns the PID of the child. In the child, `fork` returns a value of 0. Since the PID of the child is always nonzero, the return value provides an unambiguous way to tell whether the program is executing in the parent or the child.

Figure 8.13 shows a simple example of a parent process that uses `fork` to create a child process. When the `fork` call returns in line 8, `x` has a value of 1 in both the parent and child. The child increments and prints its copy of `x` in line 10. Similarly, the parent decrements and prints its copy of `x` in line 15.

code/ecf/fork.c

```
 1 #include "csapp.h"
 2 
 3 int main()
 4 {
 5     pid_t pid;
 6     int x = 1;
 7 
 8     pid = Fork();
 9     if (pid == 0) {  /* child */
10         printf("child : x=%d\n", ++x);
11         exit(0);
12     }
13 
14     /* parent */
15     printf("parent: x=%d\n", --x);
16     exit(0);
17 }
```

code/ecf/fork.c

Figure 8.13: **Using `fork` to create a new process.**

When we run the program on our Unix system, we get the following result:

```
unix> ./fork
parent: x=0
child : x=2
```

There are some subtle aspects to this simple example.

- *Call once, return twice.* The `fork` function is called once by the parent, but it returns twice: once to the parent and once to the newly created child. This is fairly straightforward for programs that create a single child. But programs with multiple instances of `fork` can be confusing and need to be reasoned about carefully.
- *Concurrent execution.* The parent and the child are separate processes that run concurrently. The instructions in their logical control flows can be interleaved by the kernel in an arbitrary way. When we run the program on our system, the parent process completes its `printf` statement first, followed by the child. However, on another system the reverse might be true. In general, as programmers we can never make assumptions about the interleaving of the instructions in different processes.
- *Duplicate but separate address spaces.* If we could halt both the parent and the child immediately after the `fork` function returned in each process, we would see that the address space of each process is identical. Each process has the same user stack, the same local variable values, the same heap, the same global variable values, and the same code. Thus, in our example program, local variable `x` has a value of 1 in both the parent and the child when the `fork` function returns in line 8. However, since the parent and the child are separate processes, they each have their own private address spaces. Any subsequent changes that a parent or child makes to `x` are private and are not reflected in the memory of the other process. This is why the variable `x` has different values in the parent and child when they call their respective `printf` statements.
- *Shared files.* When we run the example program, we notice that both parent and child print their output on the screen. The reason is that the child inherits all of the parent's open files. When the parent calls `fork`, the `stdout` file is open and directed to the screen. The child inherits this file and thus its output is also directed to the screen.

When you are first learning about the `fork` function, it is often helpful to draw a picture of the *process hierarchy*. The process hierarchy is a labeled directed graph, where each node is a process and each directed arc $a \xrightarrow{k} b$ denotes that a is the parent of b and that a created b by executing the kth lexical instance of the `fork` function in the source code.

For example, how many lines of output would the program in Figure 8.14(a) generate? Figure 8.14(b) shows the corresponding process hierarchy. The parent a creates the child b when it executes the first (and only) `fork` in the program. Both a and b call `printf` once, so the program prints two output lines.

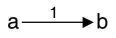
Now what if we were to call `fork` twice, as shown in Figure 8.14(c)? As we see from the process hierarchy in Figure 8.14(d), the parent a creates child b when it calls the first `fork` function. Then both a and b execute the second `fork` function, which results in the creations of c and d, for a total of four processes. Each process calls `printf`, so the program generates four output lines.

Continuing this line of thought, what would happen if we were to call `fork` three times, as in Figure 8.14(e)? As we see from the process hierarchy in Figure 8.14(f), the first `fork` creates one process, the second `fork`

```
#include "csapp.h"

int main()
{
    Fork();
    printf("hello!\n");
    exit(0);
}
```

(a) Calls `fork` once.

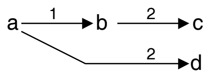


(b) Prints two output lines.

```
#include "csapp.h"

int main()
{
    Fork();
    Fork();
    printf("hello!\n");
    exit(0);
}
```

(c) Calls `fork` twice.

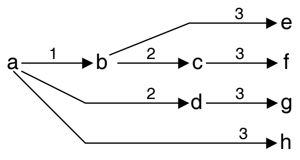


(d) Prints four output lines.

```
#include "csapp.h"

int main()
{
    Fork();
    Fork();
    Fork();
    printf("hello!\n");
    exit(0);
}
```

(e) Calls `fork` three times.



(f) Prints eight output lines.

Figure 8.14: **Examples of programs and their process hierarchies.**

creates two processes, and the third `fork` creates four processes, for a total of eight processes. Each process calls `printf`, so the program produces eight output lines.

Practice Problem 8.1:

Consider the following program:

code/ecf/forkprob0.c

```
#include "csapp.h"

int main()
{
    int x = 1;

    if (Fork() == 0)
        printf("printf1: x=%d\n", ++x);
    printf("printf2: x=%d\n", --x);
    exit(0);
}
```

code/ecf/forkprob0.c

A. What is the output of the child process?

B. What is the output of the parent process?

Practice Problem 8.2:

How many "hello" output lines does this program print?

code/ecf/forkprob1.c

```
#include "csapp.h"

int main()
{
    int i;

    for (i = 0; i < 2; i++)
        Fork();
    printf("hello!\n");
    exit(0);
}
```

code/ecf/forkprob1.c

Practice Problem 8.3:

How many "hello" output lines does this program print?

code/ecf/forkprob4.c

```
#include "csapp.h"

void doit()
{
    Fork();
    Fork();
    printf("hello\n");
    return;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

code/ecf/forkprob4.c

8.4.3 Reaping Child Processes

When a process terminates for any reason, the kernel does not remove it from the system immediately. Instead, the process is kept around in a terminated state until it is *reaped* by its parent. When the parent reaps the terminated child, the kernel passes the child's exit status to the parent, and then discards the terminated process, at which point it ceases to exist. A terminated process that has not yet been reaped is called a *zombie*.

Aside: Why are terminated children called zombies?
In folklore, a zombie is a living corpse, an entity that is half-alive and half-dead. A zombie process is similar in the sense that while it has already terminated, the kernel maintains some of its state until it can be reaped by the parent.
End Aside.

If the parent process terminates without reaping its zombie children, the kernel arranges for the `init` process to reap them. The `init` process has a PID of 1 and is created by the kernel during system initialization. Long-running programs such as shells or servers should always reap their zombie children. Even though zombies are not running, they still consume system memory resources.

A process waits for its children to terminate or stop by calling the `waitpid` function.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

returns: PID of child if OK, 0 (if WNOHANG) or -1 on error

The `waitpid` function is complicated. By default (when `options = 0`), `waitpid` suspends execution of the calling process until a child process in its *wait set* terminates. If a process in the wait set has already terminated at the time of the call, then `waitpid` returns immediately. In either case, `waitpid` returns the PID of the terminated child that caused `waitpid` to return, and the terminated child is removed from the system.

Determining the Members of the Wait Set

The members of the wait set are determined by the `pid` argument:

- If `pid > 0`, then the wait set is the singleton child process whose process ID is equal to `pid`.
- If `pid = -1`, then the wait set consists of all of the parent's child processes.

Aside: Waiting on sets of processes.
The `waitpid` function also supports other kinds of wait sets, involving Unix process groups, that we will not discuss. **End Aside.**

Modifying the Default Behavior

The default behavior can be modified by setting `options` to various combinations of the WNOHANG and WUNTRACED constants:

- WNOHANG: Return immediately (with a return value of 0) if the none of the child processes in the wait set has terminated yet.
- WUNTRACED: Suspend execution of the calling process until a process in the wait set becomes terminated or stopped. Return the PID of the terminated or stopped child that caused the return.
- WNOHANG | WUNTRACED : Suspend execution of the calling process until a child in the wait set terminates or stops, and then return the PID of the stopped or terminated child that caused the return. Also, return immediately (with a return value of 0) if none of the processes in the wait set is terminated or stopped.

Checking the Exit Status of a Reaped Child

If the `status` argument is non-NULL, then `waitpid` encodes status information about the child that caused the return in the `status` argument. The `wait.h` include file defines several macros for interpreting the `status` argument:

- WIFEXITED(status): Returns true if the child terminated normally, via a call to `exit` or a return.
- WEXITSTATUS(status): Returns the exit status of a normally terminated child. This status is only defined if WIFEXITED returned true.
- WIFSIGNALED(status): Returns true if the child process terminated because of a signal that was not caught. (Signals are explained in Section 8.5.)

- WTERMSIG(status): Returns the number of the signal that caused the child process to terminate. This status is only defined if WIFSIGNALED(status) returned true.
- WIFSTOPPED(status): Returns true if the child that caused the return is currently stopped.
- WSTOPSIG(status): Returns the number of the signal that caused the child to stop. This status is only defined if WIFSTOPPED(status) returned true.

Error Conditions

If the calling process has no children, then `waitpid` returns -1 and sets `errno` to ECHILD. If the `waitpid` function was interrupted by a signal, then it returns -1 and sets `errno` to EINTR.

Aside: Constants associated with Unix functions.
Constants such as WNOHANG and WUNTRACED are defined by system header files. For example, WNOHANG and WUNTRACED are defined (indirectly) by the `wait.h` header file:

```
/* Bits in the third argument to ‘waitpid’. */
#define WNOHANG    1   /* Don’t block waiting. */
#define WUNTRACED  2   /* Report status of stopped children. */
```

In order to use these constants, you must include the `wait.h` header file in your code:

```
#include <sys/wait.h>
```

The `man` page for each Unix function lists the header files to include whenever you use that function in your code. Also, in order to check return codes such as ECHILD and EINTR, you must include `errno.h`. To simplify our code examples, we include a single header file called `csapp.h` that includes the header files for all of the functions used in the book. The `csapp.h` header file is listed in Appendix A. **End Aside.**

Examples

Figure 8.15 shows a program that creates N children, uses `waitpid` to wait for them to terminate, and then checks the exit status of each terminated child. When we run the program on our Unix system, it produces the following output:

```
unix> ./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101
```

Notice that the program reaps the children in no particular order. Figure 8.16 shows how we might use `waitpid` to reap the children from Figure 8.15 in the same order that they were created by the parent.

Practice Problem 8.4:

Consider the following program:

code/ecf/waitprob1.c

code/ecf/waitpid1.c

```
#include "csapp.h"
#define N 2

int main()
{
    int status, i;
    pid_t pid;

    for (i = 0; i < N; i++)
        if ((pid = Fork()) == 0)  /* child */
            exit(100+i);

    /* parent waits for all of its children to terminate */
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n",
                   pid, WEXITSTATUS(status));
        else
            printf("child %d terminated abnormally\n", pid);
    }
    if (errno != ECHILD)
        unix_error("waitpid error");

    exit(0);
}
```

code/ecf/waitpid1.c

Figure 8.15: **Using the `waitpid` function to reap zombie children.**

code/ecf/waitpid2.c

```
#include "csapp.h"
#define N 2

int main()
{
    int status, i;
    pid_t pid[N+1], retpid;

    for (i = 0; i < N; i++)
        if ((pid[i] = Fork()) == 0)  /* child */
            exit(100+i);

    /* parent reaps N children in order */
    i = 0;
    while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n",
                   retpid, WEXITSTATUS(status));
        else
            printf("child %d terminated abnormally\n", retpid);
    }

    /* The only normal termination is if there are no more children */
    if (errno != ECHILD)
        unix_error("waitpid error");

    exit(0);
}
```

code/ecf/waitpid2.c

Figure 8.16: **Using `waitpid` to reap zombie children in the order they were created.**

```
#include "csapp.h"

int main()
{
    int status;
    pid_t pid;

    printf("Hello\n");
    pid = Fork();
    printf("%d\n", !pid);
    if (pid != 0) {
        if (waitpid(-1, &status, 0) > 0) {
            if (WIFEXITED(status) != 0)
                printf("%d\n", WEXITSTATUS(status));
        }
    }
    printf("Bye\n");
    exit(2);
}
```

code/ecf/waitprob1.c

A. How many output lines does this program generate?

B. What is one possible ordering of these output lines?

8.4.4 Putting Processes to Sleep

The `sleep` function suspends a process for some period of time.

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
```

returns: seconds left to sleep

`Sleep` returns zero if the requested amount of time has elapsed, and the number of seconds still left to sleep otherwise. The latter case is possible if the `sleep` function returns prematurely because it was interrupted by a *signal*. We will discuss signals in detail in Section 8.5.

Another function that we will find useful is the `pause` function, which puts the calling function to sleep until a signal is received by the process.

```
#include <unistd.h>

int pause(void);
```

always returns -1

Practice Problem 8.5:

Write a wrapper function for `sleep`, called `snooze`, with the following interface:

```
unsigned int snooze(unsigned int secs);
```

The `snooze` function behaves exactly as the `sleep` function, except that it prints a message describing how long the process actually slept. For example,

```
Slept for 4 of 5 secs.
```

8.4.5 Loading and Running Programs

The `execve` function loads and runs a new program in the context of the current process.

```
#include <unistd.h>

int execve(char *filename, char *argv[], char *envp);
```

does not return if OK, returns -1 on error

The `execve` function loads and runs the executable object file `filename` with the argument list `argv` and the environment variable list `envp`. `Execve` returns to the calling program only if there is an error such as not being able to find `filename`. So unlike `fork`, which is called once but returns twice, `execve` is called once and never returns.

The argument list is represented by the data structure shown in Figure 8.17. The `argv` variable points to

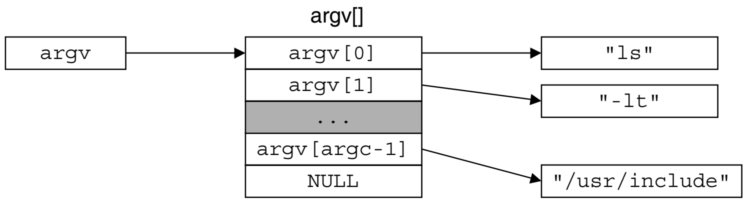


Figure 8.17: **Organization of an argument list.**

a null-terminated array of pointers, each of which points to an argument string. By convention `argv[0]` is the name of the executable object file. The list of environment variables is represented by a similar data structure, shown in Figure 8.18. The `envp` variable points to a null-terminated array of pointers to environment variable strings, each of which is a name-value pair of the form ”`NAME=VALUE`”.

After `execve` loads `filename`, it calls the startup code described in Section 7.9. The startup code sets up the stack and passes control to the main routine of the new program, which has a prototype of the form

```
int main(int argc, char **argv, char **envp);
```

or equivalently

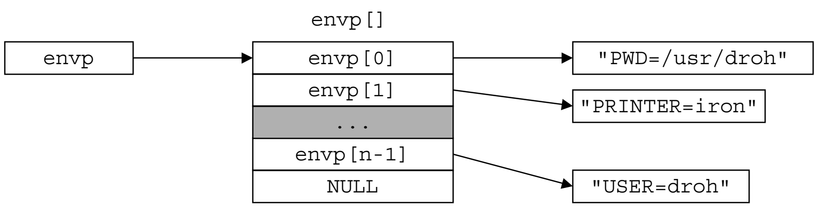


Figure 8.18: **Organization of an environment variable list.**

```
int main(int argc, char *argv[], char *envp[]);
```

When `main` begins executing on a Linux system, the user stack has the organization shown in Figure 8.19. Let's work our way from the bottom of the stack (the highest address) to the top (the lowest address). First

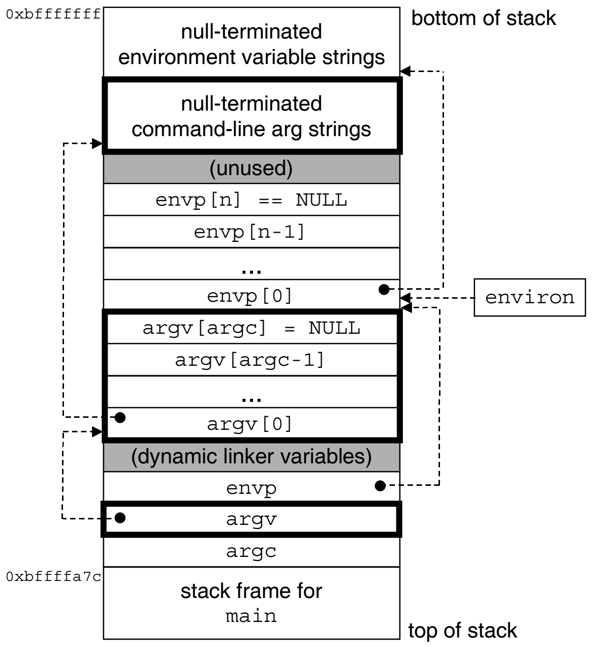


Figure 8.19: **Typical organization of the user stack when a new program starts.**

are the argument and environment strings, which are stored contiguously on the stack, one after the other without any gaps. These are followed further up the stack by a null-terminated array of pointers, each of which points to an environment variable string on the stack. The global variable `environ` points to the first of these pointers, `envp[0]`. The environment array is followed immediately by the null-terminated `argv[]` array, with each element pointing to an argument string on the stack. At the top of the stack are the three arguments to the `main` routine: (1) `envp`, which points the `envp[]` array, (2) `argv`, which points to the `argv[]` array, and (3) `argc`, which gives the number of non-null pointers in the `argv[]` array.

Unix provides several functions for manipulating the environment array.

```
#include <stdlib.h>

char *getenv(const char *name);
                                returns: ptr to name if exists, NULL if no match.
```

The `getenv` function searches the environment array for a string "`name=value`". If found, it returns a pointer to `value`, otherwise it returns `NULL`.

```
#include <stdlib.h>

int setenv(const char *name, const char *newvalue, int overwrite);
                                                     returns: 0 on success, -1 on error.
void unsetenv(const char *name);
                                                                      returns: nothing.
```

If the environment array contains a string of the form "`name=oldvalue`" then `unsetenv` deletes it and `setenv` replaces `oldvalue` with `newvalue`, but only if `overwrite` is nonzero. If `name` does not exist, then `setenv` adds "`name=newvalue`" to the array.

Aside: Setting environment variables in Solaris systems
Solaris provides the `putenv` function in place of the `setenv` function. It provides no counterpart to the `unsetenv` function. **End Aside.**

Aside: Programs vs. processes.
This is a good place to stop and make sure you understand the distinction between a program and a process. A program is a collection of code and data; programs can exist as object modules on disk or as segments in an address space. A process is a specific instance of a program in execution; a program always runs in the context of some process. Understanding this distinction is important if you want to understand the `fork` and `execve` functions. The `fork` function runs the same program in a new child process that is a duplicate of the parent. The `execve` function loads and runs a new program in the context of the current process. While it overwrites the address space of the current process, it does *not* create a new process. The new program still has the same PID, and it inherits all of the file descriptors that were open at the time of the call to the `execve` function. **End Aside.**

Practice Problem 8.6:

Write a program, called `myecho`, that prints its command line arguments and environment variables. For example:

```
unix> ./myecho arg1 arg2
Command line arguments:
    argv[ 0]: myecho
    argv[ 1]: arg1
    argv[ 2]: arg2

Environment variables:
    envp[ 0]: PWD=/usr0/droh/ics/code/ecf
    envp[ 1]: TERM=emacs
    ...
  envp[25]: USER=droh
  envp[26]: SHELL=/usr/local/bin/tcsh
  envp[27]: HOME=/usr0/droh
```

8.4.6 Using `fork` and `execve` to Run Programs

Programs such as Unix shells and Web servers (Chapter 12) make heavy use of the `fork` and `execve` functions. A shell is an interactive application-level program that runs other programs on behalf of the user. The original shell was the `sh` program, which was followed by variants such as `csh`, `tcsh`, `ksh`, and `bash`. A shell performs a sequence of *read/evaluate* steps, and then terminates. The read step reads a command line from the user. The evaluate step parses the command line and runs programs on behalf of the user.

Figure 8.20 shows the main routine of a simple shell. The shell print a command-line prompt, waits for the

code/ecf/shellex.c

```
#include "csapp.h"
#define MAXARGS   128

/* function prototypes */
void eval(char*cmdline);
int parseline(const char *cmdline, char **argv);
int builtin_command(char **argv);

int main()
{
    char cmdline[MAXLINE]; /* command line */

    while (1) {
        /* read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

code/ecf/shellex.c

Figure 8.20: **The main routine for a simple shell program.**

user to type a command line on `stdin`, and then evaluates the command line.

Figure 8.21 shows the code that evaluates the command line. Its first task is to call the `parseline` function (Figure 8.22), which parses the space-separated command-line arguments and builds the `argv` vector that will eventually be passed to `execve`. The first argument is assumed to be either the name of a built-in shell command that is interpreted immediately, or an executable object file that will be loaded and run in the context of a new child process.

If the last argument is a "&" character, then `parseline` returns 1, indicating that the program should be executed in the *background* (the shell does not wait for it to complete). Otherwise it returns 0, indicating that the program should be run in the *foreground* (the shell waits for it to complete).

After parsing the command line, the `eval` function calls the `builtin_command` function, which checks whether the first command line argument is a built-in shell command. If so, it interprets the command immediately and returns 1. Otherwise, it returns 0. Our simple shell has just one built-in command, the `quit` command, which terminates the shell. Real shells have numerous commands, such as `pwd`, `jobs`, and `fg`.

If `builtin_command` returns 0, then the shell creates a child process and executes the requested program inside the child. If the user has asked for the program to run in the background, then the shell returns to the top of the loop and waits for the next command line. Otherwise the shell uses the `waitpid` function to wait for the job to terminate. When the job terminates, the shell goes on to the next iteration.

Notice that this simple shell is flawed because it does not reap any of its background children. Correcting this flaw requires the use of signals, which we describe in the next section.

8.5 Signals

To this point in our study of exceptional control flow, we have seen how hardware and software cooperate to provide the fundamental low-level exception mechanism. We have also seen how the operating system uses exceptions to support a higher-level form of exceptional control flow known as the context switch. In this section we will study a higher-level software form of exception, known as a Unix *signal*, that allows processes to interrupt other processes.

A *signal* is a message that notifies a process that an event of some type has occurred in the system. For example, Figure 8.23 shows the 30 different types of signals that are supported on Linux systems.

Each signal type corresponds to some kind of system event. Low-level hardware exceptions are processed by the kernel's exception handlers and would not normally be visible to user processes. Signals provide a mechanism for exposing the occurrence of such exceptions to user processes. For example, if a process attempts to divide by zero, then the kernel sends it a SIGFPE signal (number 8). If a process executes an illegal instruction, the kernel sends it a SIGILL signal (number 4). If a process makes an illegal memory reference, the kernel sends it a SIGSEGV signal (number 11). Other signals correspond to higher-level software events in the kernel or in other user processes. For example, if you type a `ctrl-c` (i.e., press the `ctrl` key and the `c` key at the same time) while a process is running in the foreground, then the kernel sends a SIGINT (number 2) to the foreground process. A process can forcibly terminate another process by sending it a SIGKILL signal (number 9). When a child process terminates or stops, the kernel sends a SIGCHLD signal (number 17) to the parent.

code/ecf/shellex.c

```c
/* eval - evaluate a command line */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if (argv[0] == NULL)
        return;   /* ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        /* parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}

/* if first arg is a builtin command, run it and return true */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    if (!strcmp(argv[0], "&"))    /* ignore singleton & */
        return 1;
    return 0;                     /* not a builtin command */
}
```

code/ecf/shellex.c

Figure 8.21: `eval`**: evaluates the shell command line.**

code/ecf/shellex.c

```c
/* parseline - parse the command line and build the argv array */
int parseline(const char *cmdline, char **argv)
{
    char array[MAXLINE]; /* holds local copy of command line */
    char *buf = array;   /* ptr that traverses command line */
    char *delim;         /* points to first space delimiter */
    int argc;            /* number of args */
    int bg;              /* background job? */

    strcpy(buf, cmdline);
    buf[strlen(buf)-1] = ' ';  /* replace trailing '\n' with space */
    while (*buf && (*buf == ' ')) /* ignore leading spaces */
        buf++;

    /* build the argv list */
    argc = 0;
    while ((delim = strchr(buf, ' '))) {
        argv[argc++] = buf;
        *delim = '\0';
        buf = delim + 1;
        while (*buf && (*buf == ' ')) /* ignore spaces */
               buf++;
    }
    argv[argc] = NULL;

    if (argc == 0)  /* ignore blank line */
        return 1;

    /* should the job run in the background? */
    if ((bg = (*argv[argc-1] == '&')) != 0)
        argv[--argc] = NULL;

    return bg;
}
```

code/ecf/shellex.c

Figure 8.22: `parseline`**: parses a line of input for the shell.**

Number	Name	Default action	Corresponding event
1	SIGHUP	terminate	Terminal line hangup
2	SIGINT	terminate	Interrupt from keyboard
3	SIGQUIT	terminate	Quit from keyboard
4	SIGILL	terminate	Illegal instruction
5	SIGTRAP	terminate and dump core	Trace trap
6	SIGABRT	terminate and dump core	Abort signal from `abort` function
7	SIGBUS	terminate	Bus error
8	SIGFPE	terminate and dump core	Floating point exception
9	SIGKILL	terminate*	Kill program
10	SIGUSR1	terminate	User-defined signal 1
11	SIGSEGV	terminate and dump core	Invalid memory reference (seg fault)
12	SIGUSR2	terminate	User-defined signal 2
13	SIGPIPE	terminate	Wrote to a pipe with no reader
14	SIGALRM	terminate	Timer signal from `alarm` function
15	SIGTERM	terminate	Software termination signal
16	SIGSTKFLT	terminate	Stack fault on coprocessor
17	SIGCHLD	ignore	A child process has stopped or terminated
18	SIGCONT	ignore	Continue process if stopped
19	SIGSTOP	stop until next SIGCONT*	Stop signal not from terminal
20	SIGTSTP	stop until next SIGCONT	Stop signal from terminal
21	SIGTTIN	stop until next SIGCONT	Background process read from terminal
22	SIGTTOU	stop until next SIGCONT	Background process wrote to terminal
23	SIGURG	ignore	Urgent condition on socket
24	SIGXCPU	terminate	CPU time limit exceeded
25	SIGXFSZ	terminate	File size limit exceeded
26	SIGVTALRM	terminate	Virtual timer expired
27	SIGPROF	terminate	Profiling timer expired
28	SIGWINCH	ignore	Window size changed
29	SIGIO	terminate	I/O now possible on a descriptor.
30	SIGPWR	terminate	Power failure

Figure 8.23: **Linux signals.** Other Unix versions are similar. Notes: (1) *This signal can neither be caught nor ignored. (2) Years ago, main memory was implemented with a technology known as *core memory*. "Dumping core" is an historical term that means writing an image of the code and data memory segments to disk.

8.5.1 Signal Terminology

The transfer of a signal to a destination process occurs in two distinct steps:

- *Sending a signal.* The kernel *sends* (*delivers*) a signal to a destination process by updating some state in the context of the destination process. The signal is delivered for one of two reasons: (1) the kernel has detected a system event such as a divide-by-zero error or the termination of a child process; (2) A process has invoked the `kill` function (discussed in the next section) to explicitly request the kernel to send a signal to the destination process. A process can send a signal to itself.
- *Receiving a signal.* A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal. The process can either ignore the signal, terminate, or *catch* the signal by executing a user-level function called a *signal handler*.

A signal that has been sent but not yet received is called a *pending signal*. At any point in time, there can be at most one pending signal of a particular type. If a process has a pending signal of type k, then any subsequent signals of type k sent to that process are *not* queued; they are simply discarded. A process can selectively *block* the receipt of certain signals. When a signal is blocked, it can be delivered, but the resulting pending signal will not be received until the process unblocks the signal.

A pending signal is received at most once. For each process, the kernel maintains the set of pending signals in the `pending` bit vector, and the set of blocked signals in the `blocked` bit vector. The kernel sets bit k in `pending` whenever a signal of type k is delivered and clears bit k in `pending` whenever a signal of type k is received.

8.5.2 Sending Signals

Unix systems provide a number of mechanisms for sending signals to processes. All of the mechanisms rely on the notion of a *process group*.

Process Groups

Every process belongs to exactly one *process group*, which is identified by a positive integer *process group ID*. The `getpgrp` function returns the process group ID of the current process.

```
#include <unistd.h>

pid_t getpgrp(void);
```
returns: process group ID of calling process

By default, a child process belongs to the same process group as its parent. A process can change the process group of itself or another process by using the `setpgid` function:

```
#include <unistd.h>

pid_t setpgid(pid_t pid, pid_t pgid);
                                        returns: 0 on success, -1 on error.
```

The `setpgid` function changes the process group of process `pid` to `pgid`. If `pid` is zero, the PID of the current process is used. If `pgid` is zero, the PID of the process specified by `pid` is used for the process group ID. For example, if process 15213 is the calling process, then

```
    setpgid(0, 0);
```

creates a new process group whose process group ID is 15213, and adds process 15213 to this new group.

Sending Signals With the `kill` Program

The `/bin/kill` program sends an arbitrary signal to another process. For example

```
unix> kill -9 15213
```

sends signal 9 (SIGKILL) to process 15213. A negative PID causes the signal to be sent to every process in process group PID. For example,

```
unix> kill -9 -15213
```

sends a SIGKILL signal to every process in process group 15213.

Sending Signals From the Keyboard

Unix shells use the abstraction of a *job* to represent the processes that are created as a result of evaluating a single command line. At any point in time, there is at most one foreground job and zero or more background jobs. For example, typing

```
unix> ls | sort
```

creates a foreground job consisting of two processes connected by a Unix pipe: one running the `ls` program, the other running the `sort` program.

The shell creates a separate process group for each job. Typically, the process group ID is taken from one of the parent processes in the job. For example, Figure 8.24 shows a shell with one foreground job and two background jobs. The parent process in the foreground job has a PID of 20 and a process group ID of 20. The parent process has created two children, each of which are also members of process group 20.

Typing `ctrl-c` at the keyboard causes a SIGINT signal to be sent to the shell. The shell catches the signal (see Section 8.5.3) and then sends a SIGINT to every process in the foreground process group. In the default case, the result is to terminate the foreground job. Similarly, typing `crtl-z` sends a SIGTSTP signal to the shell, which catches it and sends a SIGTSTP signal to every process in the foreground process group. In the default case, the result is to stop (suspend) the foreground job.

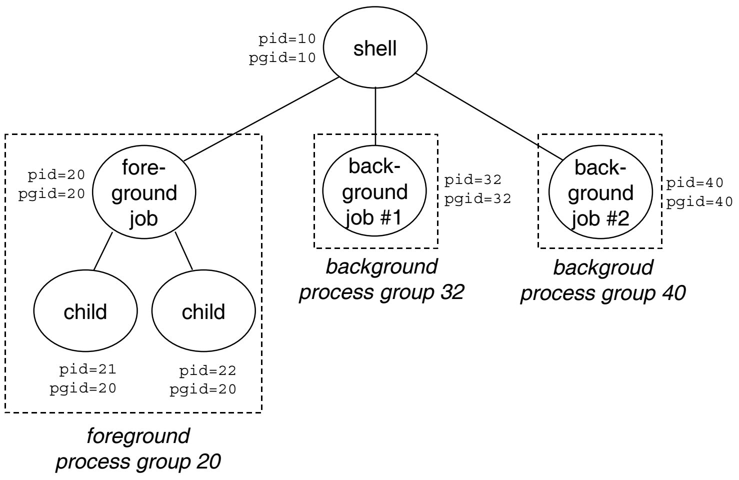


Figure 8.24: **Foreground and background process groups.**

Sending Signals With the `kill` Function

Processes send signals to other processes (including themselves) by calling the `kill` function.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```
returns: 0 if OK, -1 on error

If `pid` is greater than zero, then the `kill` function sends signal number `sig` to process `pid`. If `pid` is less than zero, than `kill` sends signal `sig` to every process in process group `abs(pid)`. Figure 8.25 shows an example of a parent that uses the `kill` function to send a SIGKILL signal to its child.

Sending Signals With the `alarm` Function

A process can send SIGALRM signals to itself by calling the `alarm` function.

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```
returns: remaining secs of previous alarm, or 0 if no previous alarm

The `alarm` function arranges for the kernel to send a SIGALRM signal to the calling process in `secs` seconds. If `secs` is zero, then no new alarm is scheduled. In any event, the call to `alarm` cancels any pending alarms, and returns the number of seconds remaining until any pending alarm was due to be delivered (had *code/ecf/kill.c*

```c
#include "csapp.h"

int main()
{
    pid_t pid;

    /* child sleeps until SIGKILL signal received, then dies */
    if ((pid = Fork()) == 0) {
        Pause();  /* wait for a signal to arrive */
        printf("control should never reach here!\n");
        exit(0);
    }

    /* parent sends a SIGKILL signal to a child */
    Kill(pid, SIGKILL);
    exit(0);
}
```

code/ecf/kill.c

Figure 8.25: **Using the `kill` function to send a signal to a child.**

not this call to `alarm` cancelled it), or 0 if there were no pending alarms.

Figure 8.26 shows a program called `alarm` that arranges to be interrupted by a SIGALRM signal every second for five seconds. When the sixth SIGALRM is delivered it terminates. When we run the program in Figure 8.26, we get the following output: a "BEEP" every second for five seconds, followed by a "BOOM" when the program terminates.

```
unix> ./alarm
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
```

Notice that the program in Figure 8.26 uses the `signal` function to install a *signal handler* function (`handler`) that is called asynchronously, interrupting the infinite `while` loop in `main`, whenever the process receives a SIGALRM signal. When the `handler` function returns, control passes back to `main`, which picks up where it was interrupted by the arrival of the signal. Installing and using signal handlers can be quite subtle, and is the topic of the next three sections.

8.5.3 Receiving Signals

When the kernel is returning from an exception handler and is ready to pass control to process p, it checks the set of unblocked pending signals (`pending & ~blocked`). If this set is empty (the usual case), then *code/ecf/alarm.c*

```
#include "csapp.h"

void handler(int sig)
{
    static int beeps = 0;

    printf("BEEP\n");
    if (++beeps < 5)
        Alarm(1); /* next SIGALRM will be delivered in 1s */
    else {
        printf("BOOM!\n");
        exit(0);
    }
}

int main()
{
    Signal(SIGALRM, handler); /* install SIGALRM handler */
    Alarm(1); /* next SIGALRM will be delivered in 1s */

    while (1) {
        ;  /* signal handler returns control here each time */
    }
    exit(0);
}
```

code/ecf/alarm.c

Figure 8.26: **Using the `alarm` function to schedule periodic events.**

the kernel passes control to the next instruction (I_{next}) in the logical control flow of p.

However, if the set is nonempty, then the kernel chooses some signal k in the set (typically the smallest k) and forces p to *receive* signal k. The receipt of the signal triggers some *action* by the process. Once the process completes the action, then control passes back to the next instruction (I_{next}) in the logical control flow of p. Each signal type has a predefined *default action*, which is one of the following:

- The process terminates.
- The process terminates and dumps core.
- The process stops until restarted by a SIGCONT signal.
- The process ignores the signal.

Figure 8.23 shows the default actions associated with each type of signal. For example, the default action for the receipt of a SIGKILL is to terminate the receiving process. On the other hand, the default action for the receipt of a SIGCHLD is to ignore the signal. A process can modify the default action associated with a signal by using the `signal` function. The only exceptions are SIGSTOP and SIGKILL, whose default actions cannot be changed.

```
#include <signal.h>

typedef void handler_t(int)

handler_t *signal(int signum, handler_t *handler)
```
returns: ptr to previous handler if OK, SIG_ERR on error (does not set errno)

The `signal` function can change the action associated with a signal `signum` in one of three ways:

- If `handler` is SIG_IGN, then signals of type `signum` are ignored.
- If `handler` is SIG_DFL, then the action for signals of type `signum` reverts to the default action.
- Otherwise, `handler` is the address of a user-defined function, called a *signal handler*, that will be called whenever the process receives a signal of type `signum`. Changing the default action by passing the address of a handler to the `signal` function is known as *installing the handler*. The invocation of the handler is called *catching the signal*. The execution of the handler is referred to as *handling the signal*.

When a process catches a signal of type k, the handler installed for signal k is invoked with a single integer argument set to k. This argument allows the same handler function to catch different types of signals.

When the handler executes its `return` statement, control (usually) passes back to the instruction in the control flow where the process was interrupted by the receipt of the signal. We say "usually" because in some systems, interrupted system calls return immediately with an error. More on this in the next section.

Figure 8.27 shows a program that catches the SIGINT signal sent by the shell whenever the user types `ctrl-c` at the keyboard. The default action for SIGINT is to immediately terminate the process. In this example, we modify the default behavior to catch the signal, print a message, and then terminate the process.

code/ecf/sigint1.c

```
 1 #include "csapp.h"
 2
 3 void handler(int sig) /* SIGINT handler */
 4 {
 5     printf("Caught SIGINT\n");
 6     exit(0);
 7 }
 8
 9 int main()
10 {
11     /* Install the SIGINT handler */
12     if (signal(SIGINT, handler) == SIG_ERR)
13         unix_error("signal error");
14
15     pause(); /* wait for the receipt of a signal */
16
17     exit(0);
18 }
```

code/ecf/sigint1.c

Figure 8.27: **A program that catches a SIGINT signal.**

The handler function is defined in lines 3–7. The main routine installs the handler in lines 12–13, and then goes to sleep until a signal is received (line 15). When the SIGINT signal is received, the handler runs, prints a message (line 5) and then terminates the process (line 6).

Practice Problem 8.7:

Write a program, called `snooze`, that takes a single command line argument, calls the `snooze` function from Problem 8.5 with this argument, and then terminates. Write your program so that the user can interrupt the snooze function by typing `ctrl-c` at the keyboard. For example:

```
unix> ./snooze 5
Slept for 3 of 5 secs.      User hits crtl-c after 3 seconds
unix>
```

8.5.4 Signal Handling Issues

Signal handling is straightforward for programs that catch a single signal and then terminate. However, subtle issues arise when a program catches multiple signals.

- *Pending signals can be blocked.* Unix signal handlers typically block pending signals of the type currently being processed by the handler. For example, suppose a process has caught a SIGINT signal and is currently running its SIGINT handler. If another SIGINT signal is sent to the process, then the SIGINT will become pending, but will not be received until after the handler returns.

- *Pending signals are not queued.* There can be at most one pending signal of any particular type. Thus, if two signals of type k are sent to a destination process while signal k is blocked because the destination process is currently executing a handler for signal k, then the second signal is simply discarded; it is not queued. The key idea is that the existence of a pending signal merely indicates that *at least* one signal has arrived.

- *System calls can be interrupted.* System calls such as `read`, `wait`, and `accept` that can potentially block the process for a long period of time are called *slow system calls*. On some systems, slow system calls that are interrupted when a handler catches a signal do not resume when the signal handler returns, but instead return immediately to the user with an error condition and `errno` set to EINTR.

Let's look more closely at the subtleties of signal handling, using a simple application that is similar in nature to real programs such as shells and Web servers. The basic structure is that a parent process creates some children that run independently for a while and then terminate. The parent must reap the children to avoid leaving zombies in the system. But we also want the parent to be free to do other work while the children are running. So we decide to reap the children with a SIGCHLD handler, instead of explicitly waiting for the children the terminate. (Recall that the kernel sends a SIGCHLD signal to the parent whenever one of its children terminates or stops.)

Figure 8.28 shows our first attempt. The parent installs a SIGCHLD handler, and then creates three children, each of which runs for 1 second and then terminates. In the meantime, the parent waits for a line of input from the terminal and then processes it. This processing is modeled by an infinite loop. When each child terminates, the kernel notifies the parent by sending it a SIGCHLD signal. The parent catches the SIGCHLD, reaps one child, does some additional cleanup work (modeled by the `sleep(2)` statement), and then returns.

The `signal1` program in Figure 8.28 seems fairly straightforward. But when we run it on our Linux system, we get the following output:

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
 <cr>
Parent processing input
```

From the output, we see that even though three SIGCHLD signals were sent to the parent, only two of these signals were received, and thus the parent only reaped two children. If we suspend the parent process, we see that indeed child process 10321 was never reaped and remains a zombie:

code/ecf/signal1.c

```
#include "csapp.h"

void handler1(int sig)
{
    pid_t pid;

    if ((pid = waitpid(-1, NULL, 0)) < 0)
        unix_error("waitpid error");
    printf("Handler reaped child %d\n", (int)pid);
    Sleep(2);
    return;
}

int main()
{
    int i, n;
    char buf[MAXBUF];

    if (signal(SIGCHLD, handler1) == SIG_ERR)
        unix_error("signal error");

    /* parent creates children */
    for (i = 0; i < 3; i++) {
        if (Fork() == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* parent waits for terminal input and then processes it */
    if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        unix_error("read");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

code/ecf/signal1.c

Figure 8.28: `signal1`: This program is flawed because it fails to deal with the facts that signals can block, signals are not queued, and system calls can be interrupted.

```
 <ctrl-z>
Suspended
linux> ps
PID TTY STAT TIME COMMAND
...
10319  p5 T    0:03 signal1
10321  p5 Z    0:00 (signal1 <zombie>)
10323  p5 R    0:00 ps
```

What went wrong? The problem is that our code failed to account for the facts that signals can block and that signals are not queued. Here's what happened:

The first signal is received and caught by the parent. While the handler is still processing the first signal, the second signal is delivered and added to the set of pending signals. However, since SIGCHLD signals are blocked by the SIGCHLD handler, the second signal is not received. Shortly thereafter, while the handler is still processing the first signal, the third signal arrives. Since there is already a pending SIGCHLD, this third SIGCHLD signal is discarded. Sometime later, after the handler has returned, the kernel notices that there is a pending SIGCHLD signal and forces the parent to receive the signal. The parent catches the signal and executes the handler a second time. After the handler finishes processing the second signal, there are no more pending SIGCHLD signals, and there never will be, because all knowledge of the third SIGCHLD has been lost. The crucial lesson is that signals cannot be used to count the occurrence of events in other processes.

To fix the problem, we must recall that the existence of a pending signal only implies that at least one signal has been delivered since the last time the process received a signal of that type. So we must modify the SIGCHLD handler to reap as many zombie children as possible each time it is invoked. Figure 8.29 shows the modified SIGCHLD handler. When we run `signal2` on our Linux system, it now correctly reaps all of the zombie children:

```
linux> ./signal2
Hello from child 10378
Hello from child 10379
Hello from child 10380
Handler reaped child 10379
Handler reaped child 10378
Handler reaped child 10380
 <cr>
Parent processing input
```

However, we are not done yet. If we run the `signal2` program on a Solaris system, it correctly reaps all of the zombie children. However, now the blocked `read` system call returns prematurely with an error, before we are able to type in our input on the keyboard:

```
solaris> ./signal2
Hello from child 18906
Hello from child 18907
Hello from child 18908
Handler reaped child 18906
Handler reaped child 18908
```

code/ecf/signal2.c

```
#include "csapp.h"

void handler2(int sig)
{
    pid_t pid;

    while ((pid = waitpid(-1, NULL, 0)) > 0)
        printf("Handler reaped child %d\n", (int)pid);
    if (errno != ECHILD)
        unix_error("waitpid error");
    Sleep(2);
    return;
}

int main()
{
    int i, n;
    char buf[MAXBUF];

    if (signal(SIGCHLD, handler2) == SIG_ERR)
        unix_error("signal error");

    /* parent creates children */
    for (i = 0; i < 3; i++) {
        if (Fork() == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* parent waits for terminal input and then processes it */
    if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        unix_error("read error");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

code/ecf/signal2.c

Figure 8.29: `signal2`**:** An improved version of Figure 8.28 that correctly accounts for the facts that signals can block and are not queued. However it does not allow for the possibility that system calls can be interrupted.

```
Handler reaped child 18907
read: Interrupted system call
```

What went wrong? The problem arises because on this particular Solaris system, slow system calls such as `read` are not restarted automatically after they are interrupted by the delivery of a signal. Instead they return prematurely to the calling application with an error condition, unlike Linux systems, which restart interrupted system calls automatically.

In order to write portable signal handling code, we must allow for the possibility that system calls will return prematurely and then restart them manually when this occurs. Figure 8.30 shows the modification to `signal1` that manually restarts aborted `read` calls. The EINTR return code in `errno` indicates that the `read` system call returned prematurely after it was interrupted.

When we run our new `signal3` program on a Solaris system, the program runs correctly:

```
solaris> ./signal3
Hello from child 19571
Hello from child 19572
Hello from child 19573
Handler reaped child 19571
Handler reaped child 19572
Handler reaped child 19573
<cr>
Parent processing input
```

8.5.5 Portable Signal Handling

The differences in signal handling semantics from system to system — such as whether or not an interrupted slow system call is restarted or aborted prematurely — is an ugly aspect of Unix signal handling. To deal with this problem, the Posix standard defines the `sigaction` function, which allows users on Posix-compliant systems such as Linux and Solaris to clearly specify the signal-handling semantics they want.

```
#include <signal.h>

int sigaction(int signum, struct sigaction *act, struct sigaction *oldact);
```
returns: 0 if OK, -1 on error

The `sigaction` function is unwieldy because it requires the user to set the entries of a structure. A cleaner approach, originally proposed by Stevens [77], is to define a wrapper function, called `Signal`, that calls `sigaction` for us. Figure 8.31 shows the definition of `Signal`, which is invoked in the same way as the `signal` function. The `Signal` wrapper installs a signal handler with the following signal-handling semantics:

- Only signals of the type currently being processed by the handler are blocked.
- As with all signal implementations, signals are not queued.

code/ecf/signal3.c

```
#include "csapp.h"

void handler2(int sig)
{
    pid_t pid;

    while ((pid = waitpid(-1, NULL, 0)) > 0)
        printf("Handler reaped child %d\n", (int)pid);
    if (errno != ECHILD)
        unix_error("waitpid error");
    Sleep(2);
    return;
}

int main() {
    int i, n;
    char buf[MAXBUF];
    pid_t pid;

    if (signal(SIGCHLD, handler2) == SIG_ERR)
        unix_error("signal error");

    /* parent creates children */
    for (i = 0; i < 3; i++) {
        pid = Fork();
        if (pid == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* Manually restart the read call if it is interrupted */
    while ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        if (errno != EINTR)
            unix_error("read error");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

code/ecf/signal3.c

Figure 8.30: `signal3`: An improved version of Figure 8.29 that correctly accounts for the fact that system calls can be interrupted.

code/src/csapp.c

```
handler_t *Signal(int signum, handler_t *handler)
{
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```

code/src/csapp.c

Figure 8.31: `Signal`: A wrapper for `sigaction` that provides portable signal handling on Posix-compliant systems.

- Interrupted system calls are automatically restarted whenever possible.
- Once the signal handler is installed, it remains installed until `Signal` is called with a `handler` argument of either SIG_IGN or SIG_DFL. (Some older Unix systems restore the signal action to its default action after a signal has been processed by a handler.)

Figure 8.32 shows a version of the `signal2` program from Figure 8.29 that uses our `Signal` wrapper to get predictable signal handling semantics on different computer systems. The only difference is that we have installed the handler with a call to `Signal` rather than a call to `signal`. The program now runs correctly on both our Solaris and Linux systems, and we no longer need to manually restart interrupted `read` system calls.

8.6 Nonlocal Jumps

C provides a form of user-level exceptional control flow, called a *nonlocal jump*, that transfers control directly from one function to another currently executing function, without having to go through the normal call-and-return sequence. Nonlocal jumps are provided by the `setjmp` and `longjmp` functions.

```
#include <setjmp.h>

int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savesigs);
```

returns: 0 from setjmp, nonzero from longjmps)

The `setjmp` function saves the current stack context in the `env` buffer, for later use by `longjmp`, and

code/ecf/signal4.c

```
#include "csapp.h"

void handler2(int sig)
{
    pid_t pid;

    while ((pid = waitpid(-1, NULL, 0)) > 0)
        printf("Handler reaped child %d\n", (int)pid);
    if (errno != ECHILD)
        unix_error("waitpid error");
    Sleep(2);
    return;
}

int main()
{
    int i, n;
    char buf[MAXBUF];
    pid_t pid;

    Signal(SIGCHLD, handler2); /* sigaction error-handling wrapper */

    /* parent creates children */
    for (i = 0; i < 3; i++) {
        pid = Fork();
        if (pid == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* parent waits for terminal input and then processes it */
    if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        unix_error("read error");

    printf("Parent processing input\n");
    while (1)
        ;
    exit(0);
}
```

code/ecf/signal4.c

Figure 8.32: `signal4`: A version of Figure 8.29 that uses our `Signal` wrapper to get portable signal-handling semantics.

returns a 0.

```
#include <setjmp.h>

void longjmp(jmp_buf env, int retval);
void siglongjmp(sigjmp_buf env, int retval);
                                                                    never returns)
```

The `longjmp` function restores the stack context from the `env` buffer and then triggers a return from the most recent `setjmp` call that initialized `env`. The `setjmp` then returns with the nonzero return value `retval`.

The interactions between `setjmp` and `longjmp` can be confusing at first glance. The `setjmp` function is called once but returns *multiple times*: once when the `setjmp` is first called and the stack context is stored in the `env` buffer, and once for each corresponding `longjmp` call. On the other hand, the `longjmp` function is called once but never returns.

An important application of nonlocal jumps is to permit an immediate return from a deeply nested function call, usually as a result of detecting some error condition. If an error condition is detected deep in a nested function call, we can use a nonlocal jump to return directly to a common localized error handler instead of laboriously unwinding the call stack.

Figure 8.33 shows an example of how this might work. The `main` routine first calls `setjmp` to save the current stack context, and then calls function `foo`, which in turn calls function `bar`. If `foo` or `bar` encounter an error, they return immediately from the `setjmp` via a `longjmp` call. The nonzero return value of the `setjmp` indicates the error type, which can then be decoded and handled in one place in the code.

Another important application of nonlocal jumps is to branch out of a signal handler to a specific code location, rather than returning to the instruction that was interrupted by the arrival of the signal. For example, if a Web server attempts to send data to a browser that has unilaterally aborted the network connection between the client and the server, (e.g., as a result of the browser's user clicking the STOP button), the kernel will send a SIGPIPE signal to the server. The default action for the SIGPIPE signal is to terminate the process, which is clearly not a good thing for a server that is supposed to run forever. Thus, a robust Web server will install a SIGPIPE handler to catch these signals. After cleaning up, the SIGPIPE handler should jump to the code that waits for the next request from a browser, rather than returning to the instruction that was interrupted by the receipt of the SIGPIPE signal. Nonlocal jumps are the only way to handle this kind of error recovery.

Figure 8.34 shows a simple program that illustrates this basic technique. The program uses signals and nonlocal jumps to do a soft restart whenever the user types `ctrl-c` at the keyboard. The `sigsetjmp` and `siglongjmp` functions are versions of `setjmp` and `longjmp` that can be used by signal handlers.

The initial call to the `sigsetjmp` function saves the stack and signal context when the program first starts. The main routine then enters an infinite processing loop. When the user types ctrl-c, the shell sends a SIGINT signal to the process, which catches it. Instead of returning from the signal handler, which would pass back control back to the interrupted processing loop, the handler performs a nonlocal jump back to the *code/ecf/setjmp.c*

```
#include "csapp.h"

jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main()
{
    int rc;

    rc = setjmp(buf);
    if (rc == 0)
        foo();
    else if (rc == 1)
        printf("Detected an error1 condition in foo\n");
    else if (rc == 2)
        printf("Detected an error2 condition in foo\n");
    else
        printf("Unknown error condition in foo\n");
    exit(0);
}

/* deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

code/ecf/setjmp.c

Figure 8.33: **Nonlocal jump example.** This example shows the framework for using nonlocal jumps to recover from error conditions in deeply nested functions without having to unwind the entire stack.

code/ecf/restart.c

```c
#include "csapp.h"

sigjmp_buf buf;

void handler(int sig)
{
    siglongjmp(buf, 1);
}

int main()
{
    Signal(SIGINT, handler);

    if (!sigsetjmp(buf, 1))
        printf("starting\n");
    else
        printf("restarting\n");

    while(1) {
        Sleep(1);
        printf("processing...\n");
    }
    exit(0);
}
```

code/ecf/restart.c

Figure 8.34: **A program that uses nonlocal jumps to restart itself when the user types `ctrl-c`.**

beginning of the `main` program.

When we ran the program on our system, we got the following output:

```
unix> ./restart
starting
processing...
processing...
restarting          user hits ctrl-c
processing...
restarting          User hits ctrl-c
processing...
```

8.7 Tools for Manipulating Processes

Unix systems provide a number of useful tools for monitoring and manipulating processes.

STRACE**:** Prints a trace of each system call invoked by a program and its children. A fascinating tool for the curious student. Compile your program with `-static` to get a cleaner trace without a lot of output related to shared libraries.

PS**:** Lists processes (including zombies) currently in the system.

TOP**:** Prints information about the resource usage of current processes.

KILL**:** Sends a signal to a process. Useful for debugging programs with signal handlers and cleaning up wayward processes.

`/proc` **(Linux and Solaris)** : A virtual filesystem that exports the contents of numerous kernel data structures in an ASCII text form that can be read by user programs. For example, type "`cat /proc/loadavg`" to see the current load average on your Linux system.

8.8 Summary

Exceptional control flow occurs at all levels of a computer system. At the hardware level, exceptions are abrupt changes in the control flow that are triggered by events in the processor. At the operating system level, the kernel triggers abrupt changes in the control flows between different processes when it performs context switches. At the interface between the operating system and applications, applications can create child processes, wait for their child processes to stop or terminate, run new programs, and catch signals from other processes. The semantics of signal handling is subtle and can vary from system to system. However, mechanisms exist on Posix-compliant systems that allow programs to clearly specify the expected signal-handling semantics. Finally, at the application level, C programs can use nonlocal jumps to bypass the normal call/return stack discipline and branch directly from one function to another.

Bibliographic Notes

The Intel macro-architecture specification contains a detailed discussion of exceptions and interrupts on Intel processors [17]. Operating systems texts [66, 71, 79] contain additional information on exceptions, processes, and signals. The classic work by Stevens [72], while somewhat outdated, remains a valuable and highly readable description of how to work with processes and signals from application programs. Bovet and Cesati give a wonderfully clear description of the Linux kernel, including details of the process and signal implementations.

Homework Problems

Homework Problem 8.8 [Category 1]:

In this chapter, we have introduced some functions with unusual call and return behaviors: `setjmp`, `longjmp`, `execve`, and `fork`. Match each function with one of the following behaviors:

A. Called once, returns twice.

B. Called once, never returns.

C. Called once, returns one or more times.

Homework Problem 8.9 [Category 1]:

What is one possible output of the following program?

code/ecf/forkprob3.c

```
#include "csapp.h"

int main()
{
    int x = 3;

    if (Fork() != 0)
        printf("x=%d\n", ++x);

    printf("x=%d\n", --x);
    exit(0);
}
```

code/ecf/forkprob3.c

Homework Problem 8.10 [Category 1]:

How many "hello" output lines does this program print?

code/ecf/forkprob5.c

```
#include "csapp.h"

void doit()
{
    if (Fork() == 0) {
        Fork();
        printf("hello\n");
        exit(0);
    }
    return;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

code/ecf/forkprob5.c

Homework Problem 8.11 [Category 1]:

How many "hello" output lines does this program print?

code/ecf/forkprob6.c

```
#include "csapp.h"

void doit()
{
    if (Fork() == 0) {
        Fork();
        printf("hello\n");
        return;
    }
    return;
}

int main()
{
    doit();
    printf("hello\n");
    exit(0);
}
```

code/ecf/forkprob6.c

Homework Problem 8.12 [Category 1]:

What is the output of the following program?

code/ecf/forkprob7.c

```
#include "csapp.h"
int counter = 1;

int main()
{
    if (fork() == 0) {
        counter--;
        exit(0);
    }
    else {
        Wait(NULL);
        printf("counter = %d\n", ++counter);
    }
    exit(0);
}
```

code/ecf/forkprob7.c

Homework Problem 8.13 [Category 1]:

Enumerate all of the possible outputs of the program in Problem 8.4.

Homework Problem 8.14 [Category 2]:

Consider the following program:

code/ecf/forkprob2.c

```
#include "csapp.h"

void end(void)
{
    printf("2");
}

int main()
{
    if (Fork() == 0)
        atexit(end);
    if (Fork() == 0)
        printf("0");
    else
        printf("1");
    exit(0);
}
```

code/ecf/forkprob2.c

Determine which of the following outputs are possible. Note: The `atexit` function takes a pointer to a function and adds it to a list of functions (initially empty) that will be called when the `exit` function is called.

A. 112002

B. 211020

C. 102120

D. 122001

E. 100212

Homework Problem 8.15 [Category 2]:

Use `execve` to write a program, called `myls`, whose behavior is identical to the `/bin/ls` program. Your program should accept the same command line arguments, interpret the identical environment variables, and produce the identical output.

The `ls` program gets the width of the screen from the COLUMNS environment variable. If COLUMNS is unset, then `ls` assumes that the screen is 80 columns wide. Thus, you can check your handling of the environment variables by setting the COLUMNS environment to something smaller than 80:

```
unix> setenv COLUMNS 40
unix> ./myls
  ...output is 40 columns wide
unix> unsetenv COLUMNS
unix> ./myls
  ...output is now 80 columns wide
```

Homework Problem 8.16 [Category 3]:

Modify the program in Figure 8.15 so that

1. Each child terminates abnormally after attempting to write to a location in the read-only text segment.
2. The parent prints output that is identical (except for the PIDs) to the following:

```
child 12255 terminated by signal 11: Segmentation fault
child 12254 terminated by signal 11: Segmentation fault
```

Hint: Read the `man` pages for `wait(2)` and `psignal(3)`.

Homework Problem 8.17 [Category 3]:

Write your own version of the Unix `system` function:

```
int mysystem(char *command);
```

The `mysystem` function executes `command` by calling "`/bin/sh -c command`", and then returns after `command` has completed. If `command` exits normally (by calling the `exit` function or executing a `return` statement), then `mysystem` returns the `command` exit status. For example, if `command` terminates by calling `exit(8)`, then `system` returns the value 8. Otherwise, if `command` terminates abnormally, then `mysystem` returns the status returned by the shell.

Homework Problem 8.18 [Category 1]:

One of your colleagues is thinking of using signals to allow a parent process to count events that occur in a child process. The idea is to notify the parent each time an event occurs by sending it a signal, and letting the parent's signal handler increment a global `counter` variable, which the parent can then inspect after the child has terminated. However, when he runs the test program in Figure 8.35 on his system, he discovers that when the parent calls `printf`, `counter` always has a value of 2, even though the child has sent five signals to the parent. Perplexed, he comes to you for help. Can you explain the bug?

code/ecf/counterprob.c

```c
#include "csapp.h"

int counter = 0;

void handler(int sig)
{
    counter++;
    sleep(1); /* do some work in the handler */
    return;
}

int main()
{
    int i;

    Signal(SIGUSR2, handler);

    if (Fork() == 0) {  /* child */
        for (i = 0; i < 5; i++) {
            Kill(getppid(), SIGUSR2);
            printf("sent SIGUSR2 to parent\n");
        }
        exit(0);
    }

    Wait(NULL);
    printf("counter=%d\n", counter);
    exit(0);
}
```

code/ecf/counterprob.c

Figure 8.35: **Counter program referenced in Problem 8.18.**

Homework Problem 8.19 [Category 3]:

Write a version of the `fgets` function, called `tfgets`, that times out after 5 seconds. The `tfgets` function accepts the same inputs as `fgets`. If the user doesn't type an input line within 5 seconds, `tfgets` returns NULL. Otherwise it returns a pointer to the input line.

Homework Problem 8.20 [Category 4]:

Using the example in Figure 8.20 as a starting point, write a shell program that supports job control. Your shell should have the following features:

- The command line typed by the user consists of a `name` and zero or more arguments, all separated by one or more spaces. If `name` is a built-in command, the shell handles it immediately and waits for the next command line. Otherwise, the shell assumes that `name` is an executable file, which it loads and runs in the context of an initial child process (job). The process group ID for the job is identical to the PID of the child.
- Each job is identified by either a process ID (PID) or a job ID (JID), which is a small arbitrary positive integer assigned by the shell. JIDs are denoted on the command line by the prefix '`%`'. For example, "`%5`" denotes JID 5, and "`5`" denotes PID 5.
- If the command line ends with an ampersand, then the shell runs the job in the background. Otherwise, the shell runs the job in the foreground.
- Typing `ctrl-c` (`ctrl-z`) causes the shell to send a SIGINT (SIGTSTP) signal to every process in the foreground process group.
- The `jobs` built-in command lists all background jobs.
- The `bg <job>` built-in command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.
- The `fg <job>` built-in command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the foreground.
- The shell reaps all of its zombie children. If any job terminates because it receives a signal that was not caught, then the shell prints a message to the terminal with the job's PID and a description of the offending signal.

Figure 8.36 shows an example shell session.

```
unix> ./shell                          Run your shell program
> bogus
bogus: Command not found.                        Execve can't find executable
> foo 10
Job 5035 terminated by signal: Interrupt    User types ctrl-c
> foo 100 &
[1] 5036 foo 100 &
> foo 200 &
[2] 5037 foo 200 &
> jobs
[1] 5036 Running    foo 100 &
[2] 5037 Running    foo 200 &
> fg %1
Job [1] 5036 stopped by signal: Stopped     User types ctrl-z
> jobs
[1] 5036 Stopped    foo 100 &
[2] 5037 Running    foo 200 &
> bg 5035
5035: No such process
> bg 5036
[1] 5036 foo 100 &
> /bin/kill 5036
Job 5036 terminated by signal: Terminated
> fg %2                                          Wait for fg job to finish.
> quit
unix>                                           Back to the Unix shell
```

Figure 8.36: **Sample shell session for Problem 8.20.**